Boston University

College of Liberal Arts

OS Support for Portable Bulk Synchronous Parallel Programs[*]

(Position Paper)

Abdelsalam Heddaya[†]
Computer Science Dept.
Boston University
heddaya@cs.bu.edu

Amr F. Fahmy[‡]
Aiken Computation Lab
Harvard University
amr@das.harvard.edu

BU-CS-94-012
December 5, 1994

Abstract

For parallel programs to become portable, they must be executable with uniform efficiency on a variety of hardware platforms, which is not the case at present. In 1990, Valiant proposed Bulk-Synchronous Parallelism (BSP) as a model on which portable parallel programs can be built [Val90a]. We argue that shared-memory BSP is efficiently implementable on a wide variety of parallel hardware, and that BSP forms a useful basis for providing an even higher level programming interface based on Sequential Consistency (SC). A list of OS memory and thread management features needed to support BSP and SC parallel programs are given, under the assumption that the parallel computer is space-shared among multiple parallel task, rather than time-shared. Known techniques to realize efficiently the most important of these features are sketched.

Report Documentation Page

Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

1. REPORT DATE **05 DEC 1994**	2. REPORT TYPE	3. DATES COVERED **05-12-1994 to 05-12-1994**

4. TITLE AND SUBTITLE **OS Support for Portable Bulk Synchronous Parallel Programs**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Defense Advanced Research Projects Agency,3701 North Fairfax Drive,Arlington,VA,22203-1714**	8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
see report

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES **10**	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**			

1 Introduction

General purpose sequential computing has enjoyed great success. This can be attributed to the existence of high level programming languages based on the von Neumann machine model [vN45] which is both simple and predictably *uniformly efficient*, in the sense that a program compiled for this model can be automatically optimized to run efficiently[1] on a variety of hardware platforms [Val90b]. The von Neumann model, formalized for the purpose of algorithm analysis as the *Random Access Machine* (RAM), is simple and "high level" because it ignores the differences in the costs of individual computational steps, as well as the variation in the cost of communication between processor and memory. Despite these simplifying assumptions, theoretical analysis of the relative performance of different von Neumann programs is robust with respect to variation in the hardware architectures for which these programs are compiled. The phenomenal success of the von Neumann model for general purpose sequential computation has been considerably amplified by the development of operating systems that facilitate the extension and sharing of processing and memory resources among disparate programs.

The story, however, has not been the same for parallel computing. The *Parallel Random Access Machine* (PRAM) model directly extends von Neumann's simplifying assumptions to also ignore the difference in the cost of memory access between local and remote memory, as well as the cost of synchronizing multiple processors. While the PRAM assumptions succeed in simplifying the theoretical analysis of parallel algorithms, it has not so far been possible to build general purpose parallel hardware on which the results of such analysis are uniformly valid and robust. In reaction, many theoreticians, as well as programmers, decided that parallel algorithms should be designed, analyzed and programmed for models that explicitly account for the detailed characteristics of the communication and synchronization hardware [Lei92]. As a result, a wide range of architectures exist today for parallel computing. Each can be programmed in its own language using its own library for its own specialized hardware. Such programs are usually hard to write in the first place and tend to be non-portable to other platforms. They lack a simple underlying programming model and thus are hard to understand. But, if we believe that general-purpose programming on a simple standard model is a pre-requisite for economical parallel computing, then we must also believe that specializing programs to detailed hardware characteristics thwarts this goal.

Several years ago, two new models have been proposed that strike a fresh balance between the

---

two extremes, Bulk-Synchronous Parallelism (BSP) [Val90a], and LogP [CKP+93]. Furthermore, at least one of these models (BSP) promises to form an excellent framework in which to implement the sequentially consistent (SC) shared memory model [Lam79] and its variants (*e.g.*, [GLL+90]). SC models share with PRAM the assumption that remote and local memory are equidistant, but do not ignore the cost of synchronization. Many parallel computer designers believe that, for large classes of SC programs that possess high locality in inter-processor communication, it is possible to achieve most of the convenience of PRAM programming, and most of the efficiency of hardware-specialized programs [LLG+92, CKA91].

In this position paper, we briefly describe the BSP and SC models, work out the OS features they require, and survey the known implementation techniques needed to deliver both a BSP *and* an SC programming interface. Throughout, we develop the intuitions underlying these models, and the constraints that any implementation must satisfy.

2 BSP and Sequential Consistency

The target hardware we are interested in modeling spans a very wide range of computation-to-communication performance ratios. Starting from one end of the scale, we find distributed memory parallel machines with specialized data and synchronization networks (*e.g.*, Cray T3D, TMC CM-5). Next, we encounter relatively generic parallel computers with no synchronization networks (*e.g.*, BBN Butterfly, IBM SP-2), and bus-based multiprocessors (*e.g.*, SGI Challenge, Sun Sparc 20). At the other, more economical and populous end of the scale, we have off-the-shelf networks of workstations (NOW). Perhaps the most interesting hardware configuration, striking a compromise between general-purposeness, cost-effectiveness, and scalability, would be *networks of (bus-based) multi-processors*. We call this latter type of system a NOMP.

Figure 1 lists the parameters needed to describe the models of shared-memory[2] BSP and SC, and to specify the tasks that the OS must perform to support these two models. A parallel machine consists of a set of p processors divided equally among $m \leq p$ nodes, each of which contains one physical memory module and one or more processors. To simplify our exposition, we set $m = p$, and assume that a processor executes one local operation per cycle. In the full paper, we will consider the case where $m < p$. A network interconnects the nodes for purposes of communication and synchronization. The *cost of communication* in terms of that of computation is captured by the

---

[2]Both the message passing and shared memory models of communication are supported by BSP.

Symbol	Term	Comments
p	Number of processors	
m	Number of memory modules	= number of multi-processor nodes. In general, $m \leq p$, with $m = p$ in the case of uniprocessor nodes.
L	Synchronization cost	in processor cycles. $L \geq \delta$, where δ = network delay, expressed also in cycles.
g	Communication cost	$g = p\times$ processor speed / network bandwidth.
v	Number of threads	$v \geq p$
$s = v/p$	Parallel slackness	$s \geq L/g$ guarantees optimal efficiency under worst case communication pattern.
d	Number of words in each memory module	Total amount of memory = $m \cdot d$ words.
$A^2 = [1..m] \times [1..d]$	Two-dimensional address space	Address $a = (i, j)$ refers to word j in node i.
$A^1 = [1..C]$	One-dimensional address space	$C > m \cdot d$ requires virtual memory.

Figure 1: **Glossary of parameters**.

parameter g, defined as the ratio between the processing rate and the network delivery rate, *i.e.*, the interval between successive words delivered by the network, expressed in processor cycles [Val90a]. In other words, g is the inverse of the per-node share of the network bandwidth, given in terms of cycles/word.

The *synchronization cost* of the machine is embodied in the parameter L, given by the number of cycles required to achieve barrier synchronization across all p processors. For a processor speed of r cycles/sec, network bandwidth of B words/sec, and network latency of δ cycles, we have $g = pr/B$ and $L \geq \delta$. This model ignores the topology of the network in the sense that all nodes are considered equidistant, and disregards any other special purpose hardware that might exist in the machine, except to the extent that it influences the values of L and g.

A parallel task consists, at any point in time, of v *threads* running on $p \leq v$ processors. Each processor is then responsible for a number of threads of the computation and the program has a *parallel slackness* of $s = v/p \geq 1$. The goal of the model is to minimize the overhead of parallelism by maximizing *efficiency* $= T_s/pT_p$, where T_p is the parallel execution time, and T_s is the best sequential execution time for the same problem instance. A program achieves portability when it can be executed on platforms with widely varying values of g and L without significant loss in efficiency. Intuitively, this is achievable only when the combined cost of communication and
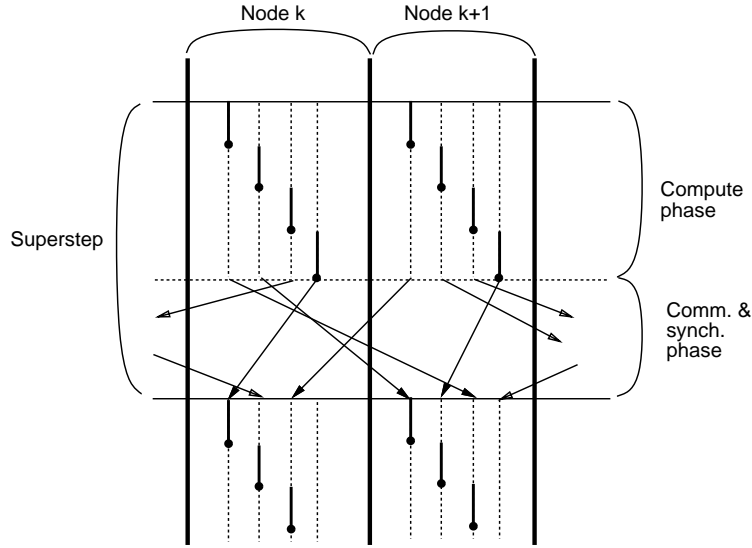
Figure 2: Structure of a BSP computation for a network of uniprocessors with $m = p$, and slack factor $s = 4$. A solid circle represents a barrier invocation, and a solid horizontal line represents its return.

synchronization does not exceed the cost of computation.

BSP. BSP stipulates a two dimensional address space A^2, where the first component of an address names a node, and the second component specifies an address in that node's local memory. A BSP computation is structured as a sequence of supersteps each followed by a barrier synchronization. A superstep is a sequence of local computations and non-local memory references, with the latter being satisfied only at the end of the superstep. Within a single superstep, a thread executes until it issues a barrier call, at which time it is suspended, and the processor switches its context from this thread to another ready thread (see figure 2). When all of the s threads running on a particular node are suspended, the node becomes idle until the network delivers to it all of the remote values it requested, and all of the updates issued by remote nodes to memory locations in its memory module. An optimally portable BSP algorithm for matrix multiplication is described in [CFSV95]. A BSP programming language and run-time library is presented in [McC94a].

SC. Sequentially consistent shared memory [Lam79] consists of a uniform cost one-dimensional address space A^1, such that memory operations performed by different threads form a global *partial*

4

order that is consistent with the ordering observed by each thread individually.[3] Intuitively, the partial ordering requirement rules out the possibility of circular data dependencies among different threads. The SC model shares with the PRAM the notion of a uniform one-dimensional address space, but departs from it in being asynchronous. The SC programmer has to write explicit synchronization statements, and hence can avoid being charged for unnecessary synchronization.

3 OS Features

The provision of two programming interfaces, one supporting BSP, and the other implementing SC, requires that the compiler, run-time system, and operating system cooperate to:

1. Allocate $p \leq vg/L$ processors to each parallel task so as to enable fair partitioning of the processors among the concurrently executing parallel tasks. The vg/L upper bound on p corresponds to an L/g lower bound on parallel slackness, which ensures that efficiency will be high even under worst case communication patterns. Intuitively, $s \geq L/g$ forces the amount of time spent in the computation phase to be at least as long as that spent in the communication and synchronization phase. Hence, processor utilization—roughly equivalent to parallel efficiency—will be at least $\frac{1}{2}$ even in the extreme case where every step in every thread requires remote communication.

   Recent measurements by McColl indicate that L/g ranges from tens to hundreds of threads per processor [McC94b].

2. Map the program's address space—be it BSP's A^2 or SC's A^1—onto the machine's distributed memory so as to avoid the creation of hot-spot modules.

3. Place threads on nodes so as to enhance locality of reference, thus minimizing communication and synchronization across the network.

4. Schedule threads so as to guarantee load balance across the machine, while keeping context switching and thread migration overhead to a minimum.

5. Automatically migrate, replicate and cache data (in conjunction with thread scheduling), so as to track and exploit shifting locality patterns.

---

[3]Sequential consistency is perhaps one of the least understood, yet most widely used shared memory models. In the full version of this paper we plan to elucidate the most common misinterpretation of SC behavior as having to agree with the global *temporal ordering* of operations on the shared memory.

6. Dynamically adjust all of the above to cope with changing sharing demands, and evolving application needs (during execution). The most compelling reason to allow this is to enable the system to run with a smaller prallel slackness when communication and synchronization requirements are low.

Virtual memory (VM) can offer additional programming convenience, for both the SC and BSP models, by allowing the address spaces A^1 and A^2 to be larger than the available physical memory. However, not all classes of applications can afford the cost of virtual memory. In [BHMW94], Burger *et al.* show that many parallel programs have working sets so large as to render VM performance overhead unacceptable.

4   Implementation Techniques

4.1   Memory Management

Mapping program data structures onto the BSP address space, A^2, is relatively straightforward: either let the programmer do it manually, or automate the most commonly used mapping functions. The Split-C programming language represents an instance of the latter approach [CDG+93]. An array X in Split-C can be automatically mapped by the compiler in *blocked* or *cyclic* fashion, corresponding to $X[i]$ being allocated in memory module $\lfloor i/p \rfloor$, or in module ($i \bmod p$), respectively. There's also a generalized *blocked cyclic* mapping for multidimensional arrays. A special iteration primitive is available that, given an array name, returns to the calling thread the array indices mapped to the same node on which the thread runs. This enables the programmer to write programs with high locality of reference, obviating the need for sophisticated run-time memory management.

Mapping of SC memory presents a more difficult problem, because it shifts the burden of discovering and enhancing data locality from the the programmer to the system. Furthermore, there is a risk that the system, in mapping address space A^1 to space A^2, may cause some memory modules to become hot-spots, by inadvertently placing popular memory locations together on the same module. Since every memory module must handle requests purely sequentially, this can create a particularly nasty form of *false sharing*. The average effect of such false sharing can be minimized, by allocating memory using a randomly chosen hash function, which minimizes the likelihood that concurrently referenced A^1 locations will fall in the same module [Val90b]. Random hashing is useful in allocating BSP memory as well because it obviates the need for randomized routing, which

6

would normally be required in order to remove communication hot-spots during the second phase of each superstep.[4]

The above technique cannot remove the hot-spots that may arise from true sharing. There are two known approaches to handle this problem. The first approach moves, copies or caches the hot-spot item, depending on the particular mix of concurrent operations issued [LEK91]. Another approach combines the concurrent operations in a tree-like structure, rooted at the hot-spot memory module. Combining can be done in hardware, as in the CM-5, or in software [Val92].

4.2   Thread Management

There are three major methods to share a machine among multiple parallel tasks: batch, time-sharing and space-sharing The first of these is the most efficient, yet the least flexible, rendering interactive parallel computing infeasible. When the OS allocates, say, a tenth of a 100-processor machine to a parallel task, it can either grant it the use of 0.1×100 processors (time-sharing), or 1×10 processors (space-sharing). Space-sharing is superior to time-sharing because it eliminates the need for context-switching among parallel tasks, and because it reduces the cost of communication and synchronization by placing more threads belonging to the same task on the same node. In terms of our model, we say that space-sharing ensures a larger parallel slackness. Thread context-switching within a single task can be made very cheap, so long as it does not require thread migration from one processor to the other.

Allocating threads to processors so as to improve locality can be achieved by delaying the binding between the thread and its data memory until after the memory has been mapped, and the location information is made available to the program (see discussion of Split-C features above). In our view, the most important problems in thread management are those of load balancing and efficient thread migration. One of the nice features of the SC memory model is that threads whose state is stored in shared memory can be migrated efficiently by copying a small amount of control state, and then relying on the memory system to fetch the remainder as needed.

---

[4]Locality of reference can be achieved despite random hashing, by following Split-C's idea of providing the names of locally resident memory objects to threads.

5 Discussion

If parallel programs are to be both portable and efficient, then a simple standard model is needed. After a brief and broad survey of the current state of the art, we sketched two models, bulk synchrony and sequential consistency, and argued for their superior portability and efficiency. We listed the OS functionality needed by BSP and SC parallel programs, and discussed the known implementation techniques that can help implement this functionality.

Networks of bus-based multi-processors hold special interest for us, for they stand the best chance of delivering the most cost-effective platform for parallel computing. In the full version of this paper, we plan to show how the BSP model can be extended straightforwardly, along the lines we sketched in section 2, to support NOMPs with ease.

The issue of locality of reference for parallel programs is a complicated one, and is not adequately addressed by section 4. A parallel program inherently possesses roughly $1/m$ less temporal locality than the corresponding sequential program, since roughly the same computation is smeared across m nodes. Spatial locality is enhanced in sequential systems by aggregating words into pages or segments, doing the same in parallel systems comes at the cost of introducing false-sharing.

Acknowledgements. Thanks to Les Valiant for useful suggestions, and to Dan Stefanescu for early discussions.

References

[BHMW94] D.C. Burger, R.S. Hyder, B.P. Miller, and D.A. Wood. Paging tradeoffs in distributed-shared-memory multiprocessors. In *Proc. Supercomputing '94*, Nov. 1994.

[CDG+93] D.E. Culler, A. Dusseau, S.C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Introduction to Split-C: Version 1.0. Technical report, Univ. of California, Berkeley, EECS, Computer Science Division, April 1993.

[CFSV95] T.E. Cheatham, A. Fahmy, D.C. Stefanescu, and L.G. Valiant. Bulk synchronous parallel computing: A paradigm for transportable software. In *Proc. 28th Hawaii International Conference on System Sciences, Maui, Hawaii*, Jan. 1995.

[CKA91] David Chaiken, John Kubiatowics, and Anant Agarwal. LimitLESS directories: a scalable cache coherence scheme. In *Proc. 4th ACM Intl. Conference on Architectural Support for Programming Languages and Operating Systems, Santa Clara, California*, pages 224–234, Apr. 1991. Describes the Alewife machine.

[CKP+93] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proc.*

4th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, San Diego, May 1993. Superseded by a later tech report.

[GLL+90] Kourosh Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multi-processors. In *Proc. 17th Annual International Symposium on Computer Architecture*, May 1990.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Computers*, C-28(9):690–691, Sep. 1979.

[Lei92] F. Thomas Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann, San Mateo, California, 1992. Vol II draft available as MIT/LCS/RSS10.

[LEK91] Richard P. LaRowe, Carla Schlatter Ellis, and Laurence S. Kaplan. The robustness of NUMA memory management. In *Proc. 13th ACM Symp. on Operating System Principles, Asilomar, California*, pages 137–151, Oct. 1991.

[LH89] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. on Computer Systems*, 7(4):321–359, Nov. 1989.

[LLG+92] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The Stanford Dash multiprocessor. *IEEE Computer*, 25(3):63–79, Mar. 1992.

[McC94a] W.F. McColl. BSP programming. In *DIMACS series of Discrete Mathematics and Theoretical Computer Science*, 1994. To appear.

[McC94b] W.F. McColl. Measurements of l, g for various parallel computing platforms. Private communication., Nov. 1994.

[Val90a] L.G. Valiant. A bridging model for parallel computation. *Comm. ACM*, 33(8):103–111, Aug. 1990.

[Val90b] L.G. Valiant. *General purpose parallel architectures.* Elsevier & MIT Press, Amsterdam, New York and Cambridge (Mass.), 1990.

[Val92] L.G. Valiant. A combining mechanism for parallel computers. Technical Report TR-24-92, Harvard University, Center for Research in Computing Technology, Nov. 1992.

[vN45] J. von Neumann. First draft of a report on the EDVAC. Technical report, University of Pennsylvania, 1945.